Tightening the Bounds on Feasible Preemption Points *

Harini Ramaprasad, Frank Mueller
Dept. of Computer Science, Center for Embedded Systems Research
North Carolina State University
Raleigh, NC 27695-7534, mueller@cs.ncsu.edu

ABSTRACT

Caches have become invaluable for higher-end architectures to hide, in part, the increasing gap between processor speed and memory access times. While the effect of caches on timing predictability of single real-time tasks has been the focus of much research, bounding the overhead of cache warm-ups after preemptions remains a challenging problem, particularly for data caches.

This paper makes multiple contributions. First, we bound the penalty of cache interference for real-time tasks by providing accurate predictions of the **data cache behavior** across preemptions, including instruction cache and pipeline effects. For every task, we derive data cache reference patterns for all scalar and non-scalar references. We show that, when considering cache preemption, the critical instance does **not** occur upon simultaneous release of all tasks.

Second, we develop analysis methods to calculate tight upper bounds on the number of possible preemption points **for each job** of a task and consider the worst-case placement of these preemption points. Partial timing of a job is performed up to a preemption point using the cache reference patterns. The effects of cache interference are then analyzed using a set-theoretic approach, which identifies the number and location of additional misses due to preemption. A feedback mechanism provides the means to interact with the timing analyzer, which subsequently times another interval of a job bounded by the next preemption.

Significant improvements in tightening bounds of up to an order of magnitude over two prior methods and up to half a magnitude over a third prior method are obtained by experiments for (a) the number of preemptions, (b) the WCET and (c) the response time of a task. Overall, this work contributes (1) by formulating a new critical instance under cache preemption, (2) by proving a new analysis method to derive bounds on the number of preemptions and (3) by determining actual preemption points when calculating the preemption delay under consideration of **data caches**.

1. INTRODUCTION

In most modern systems, data caches have become an integral part of the architecture. They provide immense savings in terms of latency. However, they have one inherent complexity, namely, the latency of memory references become unpredictable. In real-time systems, timing predictability is a central need and hence, this unpredictability due to data caches causes additional complexity of analysis.

Characterization of data cache behavior for a task is complex and has been the focus of much research. In a preemptive system, this complexity increases further. In such a system, a task with higher priority may preempt a task with lower priority at any time. This implies that some cache blocks that the lower priority task was using may now be evicted and would need to be reloaded when the task resumes execution. In recent work [19], we propose a method to bound the delay caused due to preemptions for data caches and to derive an upper bound for the response time of a task.

The issues addressed in that work are similar to those studied for instruction caches [20, 21], namely:

1. Preemption delay: Given the preempted task, the set of possible preempting tasks, and the preemption point, calculate the preemption delay that is incurred.

2. Number of preemptions: Calculate *n*, the maximum number of times a task can be preempted when it is executed as part of a given task set.

3. Worst-case scenario: Identify the placement of the *n* preemption points in the iteration space such that the worst-case total delay / preemption cost is obtained.

In this paper, we first show that the critical instance does *not* occur when all tasks are released simultaneously if we consider preemption delays. Second, we propose a new method to tightly bound the maximum number of preemptions possible for a given task. Finally, we propose a method to derive a realistic worst-case preemption scenario. The second and third contributions help us significantly tighten the WCET estimate for a task by tightening the preemption delay incurred by it.

The remainder of this paper is organized as follows. Section 3 discusses related work. Section 4 discusses the effect of considering data cache related preemption delay on the critical instance. Section 5 gives a more detailed overview of our recent work. Section 6 explains in detail the techniques used in this paper. We then present experimental results in Section 7. Section 8 summarizes the contributions of this work.

2. TASK MODEL

In our work, we consider a periodic real-time task model with period equal to the deadline of a task. The notation used in the remainder of this paper is described here. A task T_i has characteristics represented by the 7 tuple (Φ_i, P_i, C_i, c_i, B_i, R_i, $\Delta_{j,i}$). Φ_i represents the phase of the task, P_i represents the period of the task (equal to deadline), C_i represents the worst-case execution time of the task, c_i represents the best-case execution time of the task, B_i represents the blocking time of the task, R_i represents the response time of the task and $\Delta_{j,i}$ represents the preemption delay caused on the task due to a higher priority task T_j. J_{ij} represents the jth instance (job) of task T_i.

3. RELATED WORK

Several methods have been proposed in the past to bound data cache behavior for a single task without taking into account, the effects that other tasks may have on the behavior ([13], [8], [12], [25], [15]). They use methods like data flow analysis, static cache simulation, etc. for this purpose.

Analytical methods for predicting data cache behavior have been proposed. They include the Cache Miss Equations by Ghosh *et al.* [7], a probabilistic analysis method proposed by Fraguella *et al.* [6] and another analytical method by Chatterjee *et al.* [4]. The

*This work was supported in part by NSF grants CCR-0208581, CCR-0310860 and CCR-0312695.

common idea behind these methods is to characterize data cache behavior by means of a set of mathematical equations. In prior work [18], we have extended the cache miss equations framework to produce exact data cache patterns for references.

Techniques that make data caches more predictable and can be applied in preemptive systems are cache partitioning [17] and cache locking [14, 5]. Both methods lead to a significant loss in performance in order to gain predictability.

There are also several techniques that have been proposed specifically to calculate preemption delay and analyze schedulability in a multi-task preemptive system. These techniques do not specifically analyze data cache behavior, but provide a more generic solution applicable to a cache, including specific solutions for instruction caches.

Early on, Basumallick *et al.* conducted a survey of cache related issues in real-time systems [2]. This survey discussed some initial work related to the calculation of preemption delay. Busquets-Mataix *et al.* proposed a method to incorporate the effect of instruction caches on response time analysis (RTA) [3]. They compared cached RTA with cached Rate Monotonic Analysis (RMA) and concluded that cached RTA outperforms cached RMA. Lee *et al.* proposed and enhanced a method to calculate an upper bound for cache related preemption delay in a real-time system [9, 10]. They used cache states at basic block boundaries and data flow analysis on the control flow graph of a task to analyze cache behavior and calculate preemption delay.

Another approach by Tomiyama *et al.* calculates cache related preemption delay for the program path that requires the maximum number of cache blocks [22]. This path is determined by an integer linear programming technique. In this paper, an empty cache is assumed at the beginning of every job and hence, each preemption is analyzed individually. Effects of multiple preemptions are not considered. Negi *et al.* combined the techniques proposed by Tomiyama *et al.* [22] and by Lee *et al.* [9, 10] to develop an enhanced framework [16]. Once again, however, multiple preemptions are not considered in their work since an empty cache is assumed at the beginning of a task.

The work by Lee *et al.* was enhanced by Staschulat *et al.* [20, 21]. The authors propose a complete framework for the calculation of response time for tasks in a given task set. They address the three issues enumerated in the Section 1, namely calculation of the maximum number of preemption points, identification of their placement and calculation of the delay at each point. However, their focus is not on data caches, but on instruction caches.

In their work, Staschulat *et al.* discuss the concept of indirect preemptions [21]. Figure 1 illustrates the concept for a task set closely resembling their example with phase Φ, period P, WCET C and preemption delay Δ for tasks T_1 to T_4. For simplicity, Δ is assumed to be fixed per task, *i.e.*, incurred when inflicted by any higher priority task. Response times are determined as

$$R_i = C_i + B_i + \sum_{j=1..i-1} (\lceil \frac{R_i}{P_j} \rceil * C_j) + \Delta_{j,i}(R_i))$$

where the blocking time, B_i, is not considered in the example and $\Delta_{j,i}(R_i)$ is the overhead incurred by higher priority tasks preempting the current one. In Figure 1, execution is depicted by shaded boxes, the preemption delay is shown black boxes. They argue that several indirect preemptions affect lower priority tasks only once. For example, in the figure, although T_2 could be affected by every invocation of T_1, T_3 is actually only affected by the first invocation shown since, after being preempted once, it is not scheduled at all until T_2 completes execution. Thus, the response time of R_3 is 10.5 units. However, we will show in this work that the method employed by Staschulat *et al.* produces pessimistic results.

	Φ	P	C	Δ
T1	2	3	1	0
T2	1	15	5.125	0.125
T3	0	20	1.25	0.75
T4	0	25	1.25	0.25

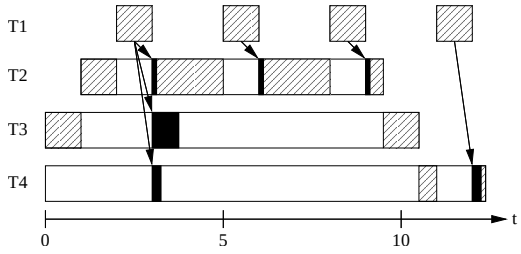


Figure 1: Preemption under WCETs, Phasing

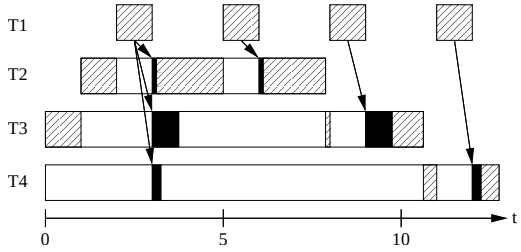


Figure 2: Preemption with Shorter Execution

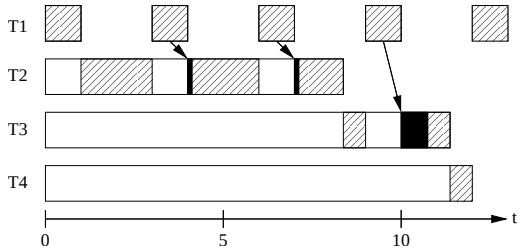


Figure 3: Preemption with WCET, $\Phi_i = 0$

4. PREEMPTION DELAY AFFECTS THE RESPONSE TIME

Prior work often assumes that the worst-case response time occurs at the theoretical critical instance for fixed-priority scheduling, *i.e.*, upon simultaneous release of all tasks. However, this is not necessarily the critical instance when preemption delays are considered. Consider Figure 3. The response time of T_3 (11.375 units) exceeds that of prior examples while the response time of T_4 (12 units) is shorter that that of Figure 2 with 12.25 units.

In general, the critical instance under preemption delay is a schedule with releases in reverse priority order such that the Φ_i of task T_i is one unit of time (one cycle) short of the preemption delay Δ_i of the same task. This theoretic result is, however, very restrictive. In practice, the hyperperiod of tasks is often a relatively small number. Hence, releases of tasks can occasionally coincide and are otherwise separated by some minimum time interval (typically 1 ms). For this reason, we consider in our work *all jobs* of a task within a hyperperiod. We calculate the number of preemptions per job and then determine the cache-related preemption delay for

the respective job and, subsequently, the response time of this job. This also enables us to consider *ranges of execution* where preemption points can occur within the code. Such job-level analysis can yield more accurate results than calculation of preemption delays per task. This helps us provide a significantly tighter estimate of the number of preemptions and, hence, the response times of jobs.

5. PRIOR WORK

In previous work [18], we enhanced a method by Vera *et al.* [23, 24] that statically analyzes data cache behavior of a single task using Cache Miss Equations [7]. This data cache analyzer was integrated into the static timing analysis framework described in prior work. [19]

The data cache analyzer produces data cache access patterns, in terms of hits and misses, for every scalar and non-scalar memory reference in a given task. It is applicable in loop nest oriented code that adheres to certain constraints as specified elsewhere [18].

These patterns give us an accurate estimate of the number of data cache misses that the task incurs and their positions in the reference stream. In this work, since we only dealt with a single task, it was sufficient to provide the number of misses instead of the actual pattern of misses and hits to the static timing analyzer described in the earlier work [18].

While the above work analyzes single tasks with respect to data caches, it does not take multi-task preemptive systems into account. In such a system, a task may be interrupted by higher priority tasks at arbitrary points during its execution. We consider non-partitioned data caches in our work. Hence, cache lines may be shared across tasks resulting in the eviction of a subset of existing memory lines from cache by preempting tasks. Assuming that all cache blocks brought in by the preempted task are evicted from cache due to preemption (*i.e.*, the cache is effectively empty after every preemption point) leads to a significant overestimation of the data cache delay. Hence, schedulability of task sets may be adversely affected so that deadlines may be missed.

In more recent work [19], we present a method to incorporate data cache delay during WCET calculation itself. Furthermore, we make the calculation of the delay as accurate as possible by considering only the intersection of the cache blocks that are useful to the preempted task once it is restarted and those that are potentially used by preempting tasks. In this work, we use response time analysis [11, 1] to determine the schedulability of a task-set. We assume a fixed-priority periodic task set where the deadline of a task is equal to its period.

The method we employ in this work has two basic phases. In the first phase, every task in a given task set is individually analyzed to produce data cache miss/hit patterns for its references. The timing analyzer is used to calculate a base WCET for every task (not including delay due to data caches).

In the second phase, the data cache analyzer and the timing analyzer interact to calculate the worst-case execution time (WCET) of the task in a multi-task preemptive system. This involves three fundamental calculations.

1. Calculation of the delay incurred by the task due to preemption at a particular point;
2. Calculation of the maximum number of possible preemptions for a given task ; and
3. Identification of the positions of these preemption points

For the second item, we calculate a pessimistic upper-bound for the number of possible preemptions. To identify preemption points and to calculate the preemption delay at a point, we use a method that involves the construction of data cache access chains.

All the data cache reference patterns of the task are merged, maintaining the order of accesses. All memory references in this consolidated pattern that access the same cache set are connected together to form a chain. Since the pattern maintains the access order, this chain accurately indicates reuse.

We identify points in the iteration space where a preemption would result in the largest cost, *i.e.*, by cutting the maximum number of distinct cache line chains. The *n* cuts with the largest cost are identified where *n* is the maximum number of preemption points incurred by the current task, as calculated in phase 1. The delays at these points are added to the WCET of the task and used in the response time analysis equations for the task set.

6. METHODOLOGY

We have described the method for calculating the WCET of a task with preemption delay in a multi-task preemptive system in Section 5. In that work, for the second and third steps, we use simplified methods that lead to overestimation of the preemption delay and, hence, the WCET of tasks.

The formula used to calculate the maximum possible number of preemptions for a task is based on the number of jobs of higher priority tasks that are released in the period of the lower priority task and the amounts of time they each take to execute. This leads to the consideration of several infeasible preemption points either because the lower priority job has not been scheduled at all and, hence, cannot be preempted, or because it has already finished executing. Further, we use the largest *n* preemption delays (where *n* is the maximum number of preemption points for the task) while calculating the WCET.

In this paper, we propose methods to calculate tight estimates of the maximum number of preemptions for a task and a safe method to identify the worst-case placement of the preemption points that is realistic.

6.1 A Tight Bound on Preemption Points

The WCET of a task is calculated with preemption delay incorporated during its calculation. Since we showed in Section 4 that the critical instance does not occur when all tasks are released at the same time, we calculate the WCET for each job of a task within a hyperperiod. Our approach handles tasks with different phases. However, in the examples in this paper, the first job of every task is assumed to be released at the same time due to current implementation constraints, which will be lifted in the future.

For the above calculation, we require the WCET and the BCET of all higher priority tasks. Further, for every task, we first calculate a base WCET that does not consider preemption delay. Since the highest priority task cannot be preempted, WCET and BCET values are calculated by simply using the static timing analyzer framework. For the other tasks, preemptions have to be considered as well.

In this section, we explain the method to eliminate infeasible preemption points without explicitly adding the preemption delay at every stage for the sake of simplicity. We discuss the calculation of preemption delay and the placement of preemption points in the iteration space of the task under consideration in the next section.

Let us discuss the methodology to eliminate infeasible preemption points through an example. Consider a three-task set with characteristics as shown in Table 1. For our calculations, we consider all jobs within a *hyperperiod*, which in this case is *200*.

The timeline for tasks T_1 and T_2 are shown in Figures 4 and 5, respectively. The arrows represent release points of higher priority

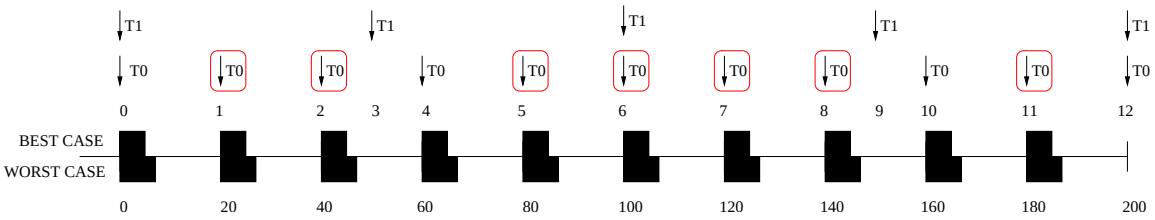


Figure 4: Timeline for Task T_1

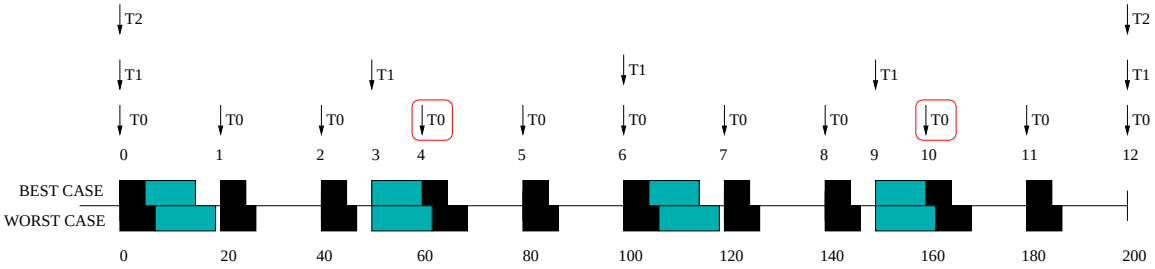


Figure 5: Timeline for Task T_2

Task	Period = deadline	WCET	BCET
T_0	20	7	5
T_1	50	12	10
T_2	200	30	25

Table 1: Task Set Characteristics

jobs and, hence, potential preemption points for the jobs of task under consideration. Preemption points are numbered consecutively. The preemption points that get eliminated by the analysis below are circled. BCETs of higher priority tasks (*e.g.*, jobs of task T_0 in the timeline for task T_1) are laid out on top, and the WCETs of higher priority tasks are below the time axis. The dark and gray rectangles show jobs of tasks T_0 and T_1 respectively.

Consider the timeline for task T_1. To check whether J_{10} can execute before preemption point 1, we use the BCET of J_{00}. Since there is idle time after placing the BCET of J_{00} (5 units), J_{10} could be scheduled before point 1. Next, we determine whether the execution of J_{10} may exceed point 1. For this purpose, we consider the sum of the WCETs of J_{00} and J_{10}, namely, 7 and 12, respectively. Since this does not exceed point 1, J_{10} is guaranteed to finish in this interval. Since J_{10} has completed execution, we determine that the maximum number of preemptions for the first job of T_1 is 0.

For the next release of T_1, *i.e.*, job J_{11}, consider the interval between preemption points 3 and 4. During this interval, in the best case, we have to consider the entire execution of the new job of T_1, namely J_{11}, that is released at point 3. Hence, for this interval,we see that job J_{11} could indeed be scheduled. Further, we see that job J_{11} is not guaranteed to finish before point 4 in the worst case. Hence, point 4 is a potential preemption point for J_{11}. Proceeding in this way, we calculate the number of preemption points for J_{11} to be 1. This example also shows that the response time for the first job (which is released at the critical instance) is not necessarily the worst possible one.

In a similar fashion, we calculate the number of preemptions for jobs of task T_2, the timeline for which is shown in Figure 5. In the case of task T_2 , there are two higher priority tasks to consider, namely T_0 and T_1.

For J_{20}, preemption point 1 is counted as a potential preemption point since there is a possibility of J_{20} being scheduled before this point, yet it does not finish before this point. For the interval between points 1 and 2, we have to consider a new job of T_0, namely J_{01}, and, in the best case, no execution of J_{10}. Hence, once again, J_{20} could be scheduled between points 1 and 2. In the worst case, we require 7 units for J_{01} during this interval. Hence, a maximum of 13 units may be used by J_{20}. Point 2 is therefore a potential preemption point for J_{20}. Proceeding this way, we consider every preemption point and test it for feasibility. We eliminate points 4 and 10 since they are not feasible preemption points for job J_{20}. Since we have reached the end of the hyperperiod of the task-set, we stop here. The upper bound for T_2, hence, is 7 preemptions. Using our original method for calculation, we obtain a bound of 9.

In summary, the method is as follows. Consider a set of tasks T_0, ..., T_n. Let J_{i0}, ..., J_{ik} represent the jobs of task T_i. Let us assume that task T_0 has the highest priority and that task T_n has the lowest priority using a static priority scheme.

For every task T_i, we construct a timeline starting from 0 up to the hyperperiod of the task-set. On this timeline, all job releases of higher priority (instances of tasks T_0 to T_{i-1}) are marked. Each of these points represents a potential preemption point for jobs of T_i.

In order to test the feasibility of a certain preemption point (say point x) for a job J_{ij}, we use the BCETs of all higher priority tasks. If the sum of these times exceeds the interval of time between points x-1 and x, the job J_{ij} has no chance of being scheduled during this interval and, hence, point x is not a feasible preemption point for J_{ij}.

If a point x is determined to be a feasible preemption point for task J_{ij}, we need to calculate the maximum time that J_{ij} can be scheduled for in the interval between x-1 and x in order to determine the remaining execution time for J_{ij}. For this purpose, to maintain safety of the analysis, we consider the sum of the WCETs of all higher priority jobs. The time remaining in the current interval after subtracting this sum, if any, is the maximum time available for J_{ij}.

Similar calculations are performed for every interval between potential preemption points until a job completes and, hence, in-

```
n: number of tasks
release_points: array of release points
timeline: array containing tasks released at every release point
interval: time interval between two preemption points
bcet_rem: array 1..n of remaining BCET (init val = 0)
wcet_rem: array 1..n of remaining WCET (init val = 0)
bcet_sum: variable to accumulate BCET within an interval
wcet_sum: variable to accumulate WCET within an interval
done, no_work_done, no_count, restart: boolean (init val = false)
current_task: task for which # preemptions is calculated
num_p: maximum number of preemptions calculated
task_num_p: array w/ maximum number of preemptions
            for every job of current task
job: task instance number (init val = 0)
t_rem: WCET of the current task

for every release point rp in {release_points}
      up to the hyper-period of the task-set {
   tasks ← timeline[release_points[rp]]
   interval ← release_points[rp+1] - release_points[rp]
   for all elements of array of tasks released at current point {
      if (element = current_task) {
         job ← job + 1
         t_rem ← wcet of current task
         restart ← true
      }
      bcet_rem[task] ← bcet[task]
      wcet_rem[task] ← wcet[task]
   }

   bcet_sum ← 0
   wcet_sum ← 0
   no_count ← false
   no_work_done ← false
   for every higher priority task hptask in task set {
      bcet_sum ← bcet_sum + bcet_rem[hptask]
      if (bcet_sum ≥ interval) {
         no_count ← true
         bcet_rem[hptask] ← bcet_sum - interval
      }
      else
         bcet_rem[hptask] ← 0
      wcet_sum ← wcet_sum + wcet_rem[hptask]
      if (wcet_sum ≥ interval) {
         wcet_rem[hptask] ← wcet_sum - interval
         no_work_done ← true
      }
      else
         wcet_rem[hptask] ← 0
   }
   if (restart = true and no_work_done = false) {
         // in the worst-case, part of curr job is executed
      if (t_rem > (interval - wcet_sum))
         t_rem ← t_rem - (interval - wcet_sum)
      else {
         t_rem = 0
         no_count ← true
      }
   if (no_count = false) // release rp is preemption point
      if (restart = true)
         num_p ← num_p + 1
   if (restart = true and t_rem = 0) { // exec of this job done
      task_num_p[job] ← num_p
      num_p ← 0
      restart ← false
   }
}
```

Figure 6: Algorithm to Eliminate Infeasible Preemption Points

feasible preemption points are eliminated. *This calculation is performed for every job within a hyperperiod.* An algorithm for this method is presented in Figure 6.

The algorithm is invoked for every task in a given task-set. It consists of a loop that iterates over all job release points in the hyperperiod of a task-set. In every iteration, we consider an interval between two preemption points. We accumulate the BCETs and WCETs of all higher priority jobs executing in this interval in the loop that traverses all higher priority tasks. Once the higher priority job executions are placed in the interval, if we find idle time in the best case, we consider the preemption point ending the interval as a potential preemption point. If we determine that the current job will not finish within the interval in the worst-case, we count the preemption point for the job under consideration. The algorithm proceeds to calculate the maximum number of feasible preemption points for every job of the current task in a hyperperiod of the task-set.

6.2 Correctness of Analysis

Consider a task set with *n* tasks, T_0, ..., T_{n-1}. Let us assume that the tasks are in decreasing order of priority. Let C_0, ..., C_{n-1} be the WCETs of the tasks and c_0, ..., c_{n-1} be their BCETs. The WCET and BCET are safe upper and lower bounds, respectively, on the longest and shortest possible execution time of a task.

Preemption of a task can only occur when it is currently *running*. Furthermore, the positions of potential preemption points for a task are fixed since they are the *release* points of a task with higher priority.

Let us consider the interval between two *consecutive* preemption points, p_{-1} and p. Let us assume that there are jobs J_{0,k_0}, ..., J_{i,k_i} have been released at some prior point and have not yet completed execution. Let us assume that J_{i,k_i} is the task for which we need to calculate the maximum number of preemptions possible.

Let *x* be the length of the interval between preemption points p_{-1} and p. We have three cases to consider.

Case 1: $\sum_{j=0}^{i-1} c_{j,k_j} < x$, $\sum_{j=0}^{i} C_{j,k_j} > x$. Assume J_{i,k_i} cannot be preempted at p, *i.e.*, it cannot be running at time p. However, $\exists_{j=0..i-1} e_{j,kj}$ s.t. $c_{j,k_j} \leq e_{j,k_j} \leq C_{j,k_j}$ and $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} < p$ and $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} > p$, *i.e.*, J_{i,k_i} is running at p. Contradiction. Hence, p is a feasible preemption point.

Case 2: $\sum_{j=0}^{i-1} c_{j,k_j} < x$, $\sum_{j=0}^{i} C_{j,k_j} < x$. Assume J_{i,k_i} can be preempted at p, *i.e.*, it may be running at time p. Hence, $\exists_{j=0..i-1} e_{j,kj}$ s.t. $c_{j,k_j} \leq e_{j,k_j} \leq C_{j,k_j}$ and $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} < p$ and $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} > p$. However, $\sum_{j=0}^{i} C_{j,k_j} < x$ implies $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} < p$. Contradiction. Hence, J_{i,k_i} cannot be running at p, and p is not a feasible preemption point.

Case 3: $\sum_{j=0}^{i-1} c_{j,k_j} > x$. Assume J_{i,k_i} can be preempted at p, *i.e.*, it may be running at time p. Hence, $\exists_{j=0..i-1} e_{j,kj}$ s.t. $c_{j,k_j} \leq e_{j,k_j} \leq C_{j,k_j}$ and $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} < p$ and $p_{-1} + \sum_{j=0}^{i} e_{j,k_j} > p$. However, $\sum_{j=0}^{i-1} c_{j,k_j} > x$ implies $p_{-1} + \sum_{j=0}^{i-1} e_{j,k_j} > p$. Contradiction. Hence, J_{i,k_i} cannot be running at p, and p is not a feasible preemption point.

Hence, preemptions can only occur under Case 1, which is the condition checked by our algorithm (see Figure 6) with the summations of WCET and BCET in the for loop and the check implemented in the subsequent conditions.

6.3 Calculation of the Preemption Delay

While the above method determines the potential preemption points, nothing has been mentioned about the *actual preemption delay that occurs at every point that is not eliminated*. This delay would, at every stage, be added to the WCET of the current task and, hence, change the amount of time remaining for the current task.

As an example, let us once again consider the task-set with characteristics shown in Table 1. Consider the interval between points 0 and 1 on the timeline for task T_2 shown in Figure 5. To calculate the delay that J_{20} incurs due to preemption at point 1, we need to identify the iteration point within J_{20} that corresponds to the time at which this preemption occurs. Since we do not know the actual execution times of the higher priority jobs (in this case, J_{00} and J_{10}), we cannot be sure of exactly by how much the execution of J_{20} proceeds in this interval. However, we may obtain upper and lower bounds for the time available for J_{20} by using the BCETs and WCETs, respectively, of higher priority tasks executing in this interval. In this example, subtracting the BCETs of J_{00} and J_{10}, namely, 5 and 10 units, from the interval time of 20 units, we get an upper bound of 5 units. The lower bound, calculated by subtracting the WCETs of J_{00} and J_{10} from the interval time, is 1 unit.

We provide each of these bounds as inputs to the static timing analyzer framework, and, for each input, we obtain two iteration points — one that represents the latest possible iteration point that may be reached in the given time (obtained from the best-case timing analysis of the task) and the other that represents the earliest iteration point that can be reached in the given time (obtained from the worst-case timing analysis of the task).

Among the four iteration points obtained above, we consider the earliest and the latest points as marking the beginning and end, respectively, of the range of iteration points that the current task could be at while it is preempted. We then choose the iteration point which would cause the highest preemption delay and take that as the worst-case delay at the preemption point being considered. This delay is added to the remainder of the execution time of the current task, and the new value is used as the remaining WCET of the current task.

In the above example, let us assume that task T_2 has a loop with 100 iterations. The static timing analyzer performs a best-case analysis and determines that, in a time interval of 1 unit (lower bound of time available for J_{20}), J_{20} can reach at most iteration 7. By performing worst-case analysis, it determines that J_{20} is sure to reach at least iteration 4 in 1 time unit. Similarly, it determines that J_{20} can reach at most iteration 13 and at least iteration 9 in 5 time units (upper bound of time available for J_{20}). Hence, the range of iteration points that we need to consider is 4 to 13. Among these iteration points, we choose the one that would produce highest preemption delay and add this delay to the remaining execution time of J_{20}.

An algorithm for the calculation of the WCET bound by repeated interaction with the static timing analyzer, depicted in Figure 7, is described below. For every job, the preemption delays at every point in the access chains is first calculated. Next, the algorithm to calculate the number of preemptions for the current job is invoked. The timing analyzer is then invoked to get the range of iteration points that need to be considered for calculation of delay at a given preemption point and the maximum delay in the given range of iteration points is added to the WCET of the current job. This process, starting from the calculation of preemption delays for points in the access chain, is repeated for the next preemption point until there are no more preemption points to consider.

```
curr_job: current job being considered
done: bool // calculation of WCET complete?
curr_preempt_index: current preemption point considered
chain_info: array of delays incurred due to preemption
      at every point in the access chain
max_preempts: max. # preemptions for current job
min_iter_pt: earliest iteration point (IP) reached by
      curr_job in a given time
max_iter_pt: latest IP reached by curr_job
      in a given time
min_exec_time: BCET of curr_job
max_exec_time: WCET of curr_job
wcet: array containing WCET of every job of a task
max_delay: preemption delay calculated at every stage

curr_preempt_index ← 0
while (done = false)
{
   chain_info ← calcAccessChainWeights(curr_job)
   max_preempts ← calcMaxNumOfPreemptions(curr_job)
   min_iter_pt ← getMinIterationPoint(min_exec_time)
   max_iter_pt ← getMaxIterationPoint(max_exec_time)
   max_delay ← max_delay + calcWCDelayInRange(min_iter_pt,
                        max_iter_pt, chain_info, curr_job)
   if (curr_preempt_index ≥ max_preempts)
      done ← true
   wcet[curr_job] ← wcet[curr_job] + max_delay
}
```

Figure 7: Bound WCET + Preemption Delays

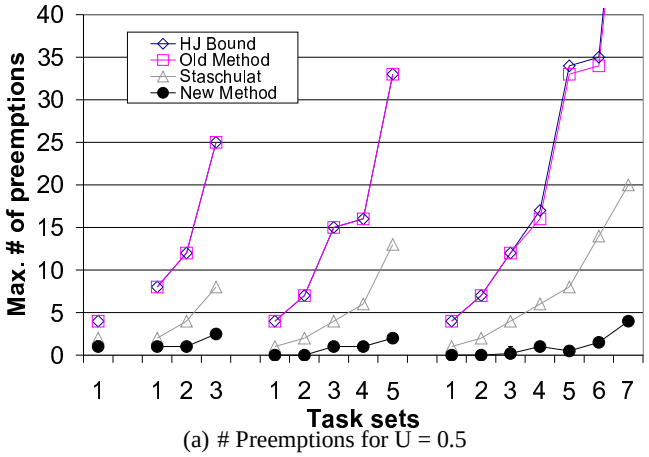
6.4 Complexity of Analysis

For every task, the single task analysis is performed only once. In this analysis, we walk through the iteration space of the task in order to calculate the number and positions of data cache misses. Hence, the time and space complexity for every task is $O(n)$ where *n* is the number of data references of the task.

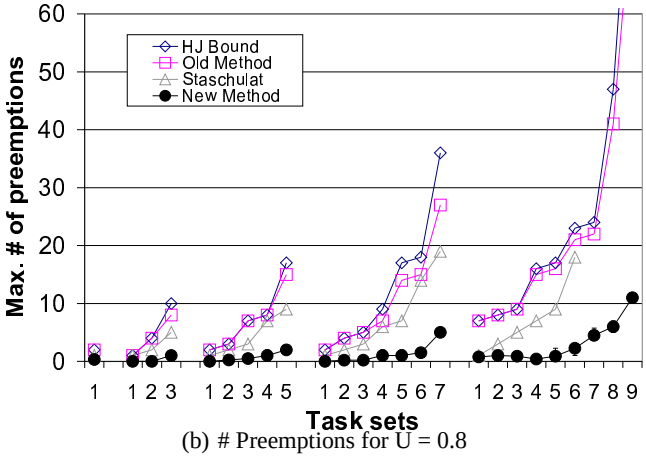
To calculate worst-case preemption delay, first of all, we consider every job of a task within the hyperperiod separately. For every job, calculation of the number of preemption points has a time complexity of $O(hp)$ where *hp* is the number of higher priority jobs released during the period of the job since we have to test every point for feasibility. For calculation of preemption delay at a point, we have a constant time complexity. This is because we simply look up the weight of the access chain for the given job at the given point. In order to identify the worst-case placement of a given preemption point in the access space of a job, the time complexity is $O(n)$, *i.e.*, we iterate over the range of possible iteration points identified as potential positions for the preemption point (explained in Section 6.3) in order to find the one with maximum preemption cost. This is usually a small range since the maximum possible execution time for a job within an interval is no larger than the largest interval between potential preemption points.

7. RESULTS

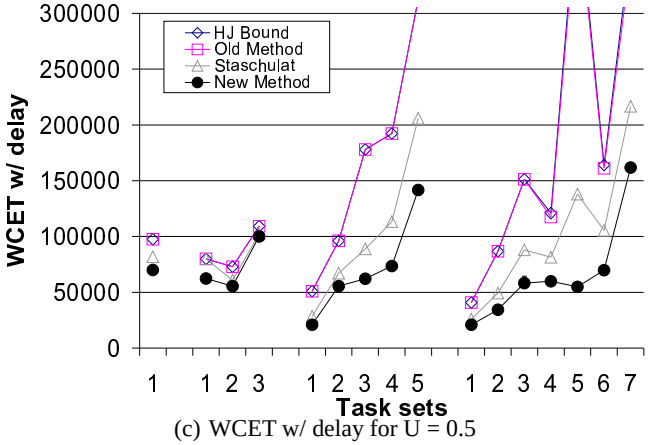
In all our experiments, we use benchmarks from the DSPStone benchmark suite [26], the details of which are described in earlier work [19]. We conducted experiments with several task sets constructed using the DSPStone benchmarks with different data set sizes. We used tasks sets that have a base utilization (utilization



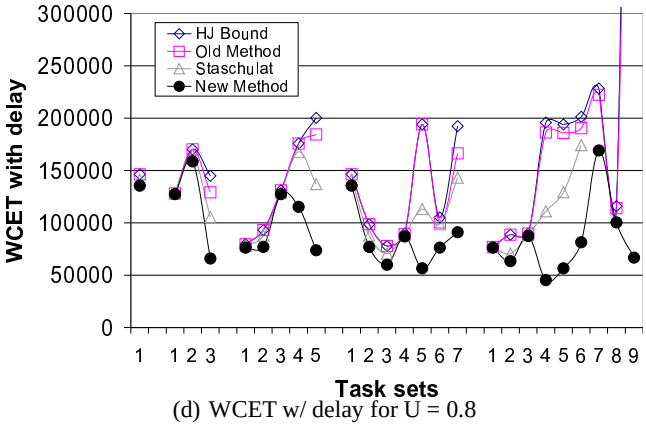
(a) # Preemptions for U = 0.5



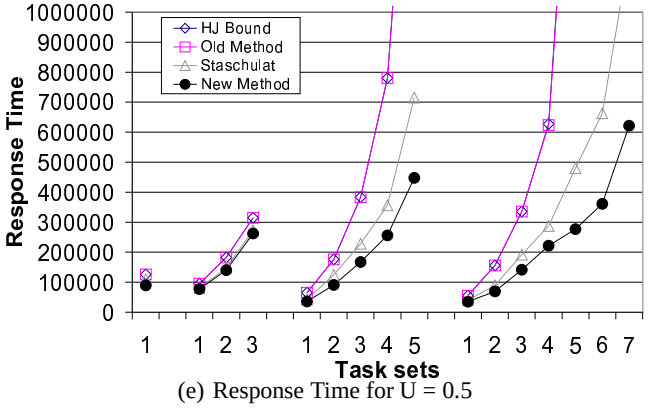
(b) # Preemptions for U = 0.8



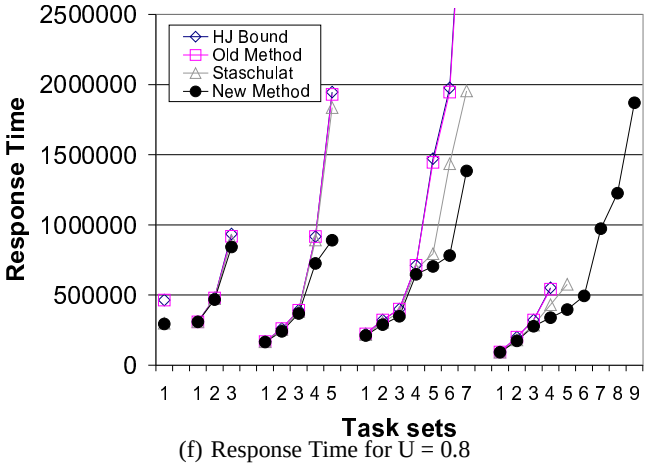
(c) WCET w/ delay for U = 0.5



(d) WCET w/ delay for U = 0.8



(e) Response Time for U = 0.5



(f) Response Time for U = 0.8

Figure 8: Experimental Results for Utilizations of 0.5 and 0.8

Benchmark	Period	WCET	BCET	# Jobs	# Preempts New Method			# Preempts HJ Bound	# Preempts Old Method	# Preempts Staschulat
					avg	min	max			
200convolution	100000	14191	14191	40	**0**	**0**	**0**	0	0	0
300convolution	400000	20891	20891	10	**0**	**0**	**0**	4	4	1
500convolution	500000	34291	34291	8	**0**	**0**	**0**	7	7	2
300n-real-updates	800000	56538	47338	5	**0.2**	**0**	**1**	12	12	4
matrix1	1000000	59896	54015	4	**1**	**1**	**1**	17	16	6
600fir	2000000	54837	52537	2	**0.5**	**0**	**1**	34	33	8
800convolution	2000000	66191	54391	2	**1.5**	**1**	**2**	35	34	14
900lms	4000000	158636	118536	1	**4**	**4**	**4**	71	67	20

Table 2: Preemptions for Taskset with U=0.5

Benchmark	Period	WCET	BCET	# Jobs	# Preempts New Method			# Preempts HJ Bound	# Preempts Old Method	# Preempts Staschulat
					avg	min	max			
n-real-updates	100000	16738	16838	50	**0**	**0**	**0**	0	0	0
900convolution	625000	76391	61091	8	**0.75**	**0**	**1**	7	7	1
matrix1	625000	59896	54015	8	**1**	**1**	**1**	8	8	3
1000convolution	625000	87091	67791	8	**0.875**	**1**	**2**	9	9	5
600convolution	1000000	45291	40991	5	**0.4**	**0**	**1**	16	15	7
300n-real-updates	1000000	56538	47338	5	**0.875**	**1**	**3**	17	16	9
800fir	1250000	77037	69737	4	**2.25**	**1**	**3**	23	21	18
900lms	1250000	158636	118536	4	**4.5**	**5**	**7**	24	22	
1000fir	2500000	99237	86937	2	**6**	**5**	**7**	47	41	
500fir	5000000	43937	43937	1	**11**	**11**	**11**	94	80	

Table 3: Preemptions for Taskset with U=0.8

without considering preemption delays) of 0.5, 0.6, 0.7 and 0.8. For each of these utilization values, we constructed task sets with 2, 4, 6 and 8 tasks. We also constructed a set with 10 tasks for 0.8 utilization.

For the sake of comparison, we calculate the maximum number of preemptions (n) possible for a task using four different methods.

1. A higher-priority job bound (HJ Bound) is determined by simply counting the number of higher priority jobs for a task. This method uses only the periods of tasks.

2. We calculate a tighter bound for the number of preemptions using the old method proposed in prior work [19]. This method uses the periods and WCETs of tasks.

3. We calculate the maximum number of methods considering indirect preemption effects as proposed by Staschulat *et al*. This method uses the periods and response times of tasks. [21].

4. We calculate the maximum number of preemptions using the range of execution times of higher priority jobs as proposed in this paper. This method uses the periods, WCETs and BCETs of tasks.

The first three methods of calculating the number of preemptions give us no idea about the actual placement of the preemption points. Hence, we aggregate the maximum n delays possible for the given task to obtain the worst-case data cache related preemption delay.

We present results of complete response time analysis for task-sets using real benchmarks. The results of the experiments for utilizations 0.5 and 0.8 are presented in the graphs shown in Figure 8. Results for 0.6 and 0.7 are similar and have been omitted due to space constraints. Each graph shows a different metric, WCET with preemption delay, response time and maximum number of preemptions, given a certain base utilization. The x-axis shows the various task sets with 2, 4, 6 and 8 tasks. The plots include all tasks except the highest priority task in every task set, which was omitted since it cannot be preempted.

In all our results, we see that our new method derives a much tighter estimate of the maximum number of preemptions for a task and, hence, significantly tighter estimates of the WCET with delay and the response time of a task. In some of the results, the methods used as comparison do not have response time values in the graph (*e.g.,* task sets 3, 4 and 5 for 0.8 base utilization). This means that the response time was, in those cases, greater than the period, hence making the task set unschedulable. Our method shows that, in reality, those task sets are schedulable. This underlines the potentially significant benefit of our new method. Further, in the case of the method proposed by Staschulat *et al.*, we calculate the maximum number of preemptions for a task based on its response time. Hence, if the response time turns out to be greater than the period, we do not report the value for maximum number of preemptions for the task by this method.

We also observe that, within a task set, as we proceed towards lower priority tasks, our method's effectiveness improves. This is indicated by the widening gap between the other methods and our new method as we go towards lower priority tasks (up to an order of magnitude). This is because the lower priority tasks have much lower chances of getting scheduled in the initial intervals between preemptions points. Hence, more preemption points are deemed infeasible by our method, thus tightening the bounds of the metrics.

Task ID	Period	WCET	# Preempts New Method (Min/Max/Avg)					# Preempts HJ Bound	# Preempts Old Method	# Preempts Staschulat
			W/B = 1	W/B = 1.5	W/B = 2	W/B = 2.5	W/B = 3			
1	810000	16000	1/1/1	1/1/1	1/1/1	1/1/1	1/1/1	8	8	2
2	100000	5000	0/1/0.25	0/1/0.25	0/2/0.5	0/2/0.5	0/2/0.5	12	12	4
3	200000	30000	3/3/3	3/4/3.5	3/5/4	3/5/4	3/5/4	25	25	8

Table 4: Preemptions for Taskset with U=0.5 for Varying WCET/BCET (W/B) rations

Task ID	Period	WCET	# Preempts New Method (Min/Max/Avg)					# Preempts HJ Bound	# Preempts Old Method	# Preempts Staschulat
			W/B = 1	W/B = 1.5	W/B = 2	W/B = 2.5	W/B = 3			
1	80000	20000	2/2/2	2/2/2	2/2/2	2/2/2	2/2/2	8	8	3
2	100000	12000	1/2/1.5	1/3/1.75	1/3/1.75	1/4/2	1/4/2	12	12	6
3	200000	50000	6/6/6	7/10/8.5	8/12/10	9/14/11.5	9/14/11.5	25	25	19

Table 5: Preemptions for Taskset with U=0.8 for Varying WCET/BCET (W/B) rations

The results with utilization 0.8 show a higher number of preemptions than the one with utilization 0.5. At the higher utilization, some tasks have a higher WCET and, hence, can be preempted more frequently. Due to the increased number of preemptions, we also observe higher response times in this case.

Notice that the priority of a task is not significant in terms of its WCET bound, even when including the preemption delay, mostly because the base WCET dominates the preemption delay cost. This is evident for task 6 in Figures 8(c) and 8(d), which has a lower WCET with delay than its predecessor, task 5. In other words, the ordering of tasks is rate-monotone, not necessary WCET-monotone.

From the results, we make several observations about prior methods. For the task with second highest priority in each task set, since there is only one task above it, we observe that the HJ bound, our old method and the method proposed by Staschulat *et al.* give the same result. However, as we proceed towards lower priority tasks within a task set, our old method gives tighter results when compared to the HJ bound. This is because our old method takes into account the WCET of a task and not just the period as the HJ bound method does. The method proposed by Staschulat *et al.* produces tighter results when compared to both our old method and the HJ bound. This is because the Staschulat method considers the effects of indirect preemptions correctly. However, the new method proposed in this paper produces tighter results than all three prior methods.

In order to show the variation in the maximum number of preemptions obtained by our new method between the various jobs of a task, we provide results for two task sets of different sizes in Tables 2 and 3.

We observe that the new method always produces a *significantly* lower value than that produced by the previous methods. As we proceed towards lower priority tasks, we observe differences in the minimum, maximum and average number of preemptions for different jobs. Further, it was observed during experimentation (not indicated in tables) that the maximum value for number of preemptions was not always obtained for the first job of the task (released at the same time as all higher priority jobs). This proves the claim we make in Section 4 about the critical instance *not* being the instance at which jobs of all tasks are released at the same time. Here again, in the case of 0.8 utilization, we do not report the maximum number of preemptions obtained by the Staschulat method for some tasks. This is because the task has a response time that is greater than its period and, hence, we cannot calculate the maximum number of preemptions, which is based on the response time.

Finally, we performed a series of experiments with synthetic task sets where we vary the ratio of the WCET of a task to its BCET, maintaining all other parameters. The results of these experiments for utilizations 0.5 and 0.8 are shown in Tables 4 and 5, respectively. We obtained results for ratios of 1, 1.5, 2, 2.5 and 3 for each of the utilizations. The results indicate that the number of preemptions calculated by our are significantly lower than for previous methods. Furthermore, for our new method this metric only varies for low values of the WCET/BCET ratio. Ratios of 3 or higher settle at a fixpoint for this task set, *i.e.*, if the BCET decreases any further, it does not affect our calculation of the maximum number of preemptions. Hence, we could calculate maximum number of preemptions for various WCET/BCET ratios for a given task set. Alternatively, if the preemption bound saturates at a low ratio, there is no need to calculate the BCET for a task at all. Instead, we could use a value of BCET=0.

8. CONCLUSION

This work provides a method to calculate cache related preemption delay. It is specifically suited to data caches and integrates with past would in instruction cache and pipeline analysis. We enhance a framework that was developed in prior work to calculate bounds for the preemption delay within data cache reference patterns for real-time tasks. Using these bounds to calculate tighter estimates of the WCET of tasks, we perform response time analysis on all tasks in a task set to determine its schedulability.

The contributions of this paper are:

1. Determination of a new critical instance under cache preemption;
2. calculation of a significantly tighter bound for the maximum number of preemptions possible for a given task; and
3. construction of a realistic worst-case scenario for the placement of preemption points.

A feedback mechanism provides the means to interact with the timing analyzer, which subsequently times another interval of a task bounded by the next preemption.

Our results show that a significant improvement (of up to an order of magnitude over some prior methods and up to half an order of magnitude over others) in bounds for (a) the number of preemptions, (b) the WCET and (c) the response time of a task are obtained. This work also contributes a methodology to integrate data

caches into preemption delay determination under response-time analysis and, in this context, considers a critical instances of staggered releases, both of which are novel, to the best of our knowledge. Future work will quantify the effect of phasing on bounding feasible preemption points.

9. REFERENCES

[1] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[2] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[3] J. V. Busquets-Matraix. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In *EuroMicro Workshop on Real-Time Systems*, June 1996.

[4] S. Chatterjee, E. Parker, P. Hanlon, and A. Lebeck. Exact analysis of the cache behavior of nested loops. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–297, June 2001.

[5] D. Decotigny and I. Puaut. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, page 114, dec 2002.

[6] B. B. Fraguela, R. Doallo, and E. L. Zapata. Automatic analytical modeling for the estimation of cache misses. In *International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[7] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, 1999.

[8] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Embedded Technology and Applications Symposium*, June 1996.

[9] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis or cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[10] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, Nov. 2001.

[11] J. Lehoczky, L. Sha, , and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the Real-Time Systems Symposium*, Santa Monica, California, Dec. 1989.

[12] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, Dec. 1996.

[13] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, Dec. 1994.

[14] B. Lisper and X. Vera. Data cache locking for higher program predictability. In *ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–282, Mar. 06 2003.

[15] T. Lundqvist and P. Stenstrm. Empirical bounds on data caching in high-performance real-time systems. Technical report, Chalmers University of Technology, 1999.

[16] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. *ACM International Symposium on Hardware Software Codesign*, Oct. 2003.

[17] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, 2002.

[18] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 148–157, Mar. 2005.

[19] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *IEEE Real-Time Embedded Technology and Applications Symposium*, page (accepted), Apr. 2006.

[20] J. Staschulat and R. Ernst. Multiple process execution in cache related preemption delay analysis. In *ACM International Conference on Embedded Software*, 2004.

[21] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Euromicro Conference on Real-Time Systems*, 2005.

[22] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. *ACM International Symposium on Hardware Software Codesign*, 2000.

[23] X. Vera, J. Llosa, A. González, and N. Bermudo. A fast and accurate approach to analyze cache memory behavior (research note). *Lecture Notes in Computer Science*, 1900:194–198, 2000.

[24] X. Vera and J. Xue. Let's study whole-program cache behavior analytically. In *International Symposium on High Performance Computer Architecture*. IEEE, Feb. 2002.

[25] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, Nov. 1999.

[26] V. Zivojnovic, J. Velarde, C. Schlager, and H. Meyr. Dspstone: A dsp-oriented benchmarking methodology. In *Signal Processing Applications and Technology*, 1994.